OpenCL JIT Compilation for Dynamic Programming Languages

OpenCL JIT Compilation for Dynamic Programming Languages

Juan Fumero[†] Michel Steuwer[†] Lukas Stadler[*] Christophe Dubach[†]

[†]The University of Edinburgh, [*]Oracle Labs, AT

juan.fumero@ed.ac.uk michel.steuwer@ed.ac.uk lukas.stadler@oracle.com christophe.dubach@ed.ac.uk

Abstract

Graphics Processor Units (GPUs) are powerful hardware to parallelize and speed-up applications. However, programming these devices is too complex for most users and the existing standards for GPU programming are available only for low-level languages such as C.

Dynamic programming languages offer higher abstractions and functionality for many users. GPU programming is possible for dynamic languages through external libraries or via wrappers in which the GPU code is normally written in C. Either way, programmers have to rely on third-party libraries with a limited number of operations or program the GPU kernels themselves.

In this work we present a technique to automatically offload parts of the input program written in a dynamic language into OpenCL without any changes in the original source code. Our preliminary results show we achieve speedups of up to 150x when using the GPU.

1. Motivation

Despite their popularity, GPUs remain very hard to program. GPUs are normally programmed with low-level languages similar to C with GPU-specific extensions which makes the programmability of these devices too complex for most users. Good understanding of the GPU architecture is a requirement to achieve high performance. This includes understanding the GPU memory hierarchy or threads organization and scheduling. This is a challenging task, especially for non-expert GPU programmers.

Scientist and non-expert programmers prefer to use higher-level programming languages such as R, Ruby or Javascript even if they perform slowly. These programming languages are simpler and easier to use, enabling faster prototyping and software development. GPU programming for dynamic languages is available through external libraries, which normally contain a set of fixed operations to execute on GPUs, or via wrappers, in which the programmers write the GPU kernel in C language and then compiled dynamically.

However, there is no compiler that dynamically offloads generic input programs written in a high-level dynamic programming language to the GPU. One of the reasons is the difficulty of implementing a Just-In-Time (JIT) compiler. This task already requires a lot of engineering effort on traditional CPUs. But when running on GPUs, this is even more complicated because of limited high-level programmability features of the GPUs compared to the CPUs.

GPUs can only execute a subset of C code, with no support for exceptions, traps, pointers, recursion and dynamic memory allocation. In dynamic programming languages, a simple operation such as `a + b` can perform many runtime checks such as type checking, null checking (if any of the operands is `null`) or checks if extra allocations are needed. All of these constrains have to be taking into account when generating GPU programs.

Partial evaluation is a compiler technique to improve the performance of dynamic and interpreted programming languages. This technique has been demonstrated for CPUs (Würthinger et al.). Partial evaluation uses profiling information to specialise the input program with the observed types, resulting in significant reduction of the interpreter overhead.

In this work we propose to use partial evaluation for GPU compilation. Based on our prior work (Fumero et al.) we show how to generate OpenCL C code for the R language and we propose an extension to multiple dynamic languages. We present an overview of how to dynamically compile R and Ruby programs to OpenCL. We extended the existing FastR (Stadler et al.) R interpreter and JRuby both built on top of Truffle framework (Würthinger et al.), a compiler infrastructure to build languages on top of the JVM, and Graal (Duboscq et al.), a novel Java JIT Compiler.

2. OpenCL JIT Compilation

This sections gives an overview of how to generate and execute OpenCL code from dynamic programming languages using Truffle and Graal as main compiler infrastructure. We show how the OpenCL JIT compilation is performed for R and Ruby programming languages.

2.1 Identify the parallelism

First, we need to identify if an input code is parallel or not. To do that, we take advantage of the existing functions in the language that operate on arrays, as skeletons for generating parallel code. For instance, the R language contains a set

MoreVMs'17 April 3, 2017, Brussels, Belgium

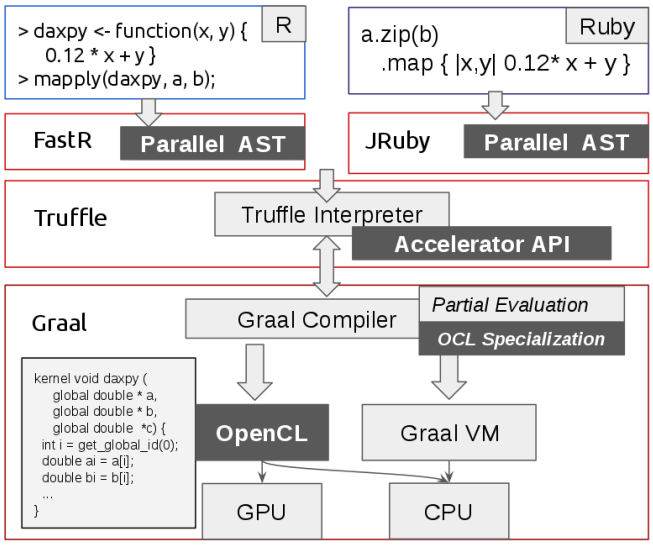

Figure 1. Compiler software stack for R and Ruby language to generate OpenCL C code from parallel skeletons such as the R `mapply` and Ruby `map` functions.

of functions called `apply`, which execute an input function for every input element of an array. In the case of Ruby, the language contains the `map` primitive that is equivalent to the R's `apply` function. Both of these functions can be easily executed in parallel.

Figure 1 shows an overview of our compiler stack for compiling R and Ruby programs to OpenCL. The top of the figure shows an example (daxpy) written in R (left side) and Ruby (right side). The dark squares show our contributions to the existing compiler infrastructure. We first parse the input function into an Abstract Syntax Tree (AST). Each language has each own AST interpreter implemented with Truffe. We identify the functions such as the `apply` in R or the `map` in Ruby, as potential parallel operations and we represent them as a new node in the AST.

2.2 AST Interpreter Execution

Once the AST is built, we execute the program in the AST interpreter. We first execute the AST interpreter on the CPU sequentially to specialize it to the observed data types and remove as much interpreter overhead as possible. This allow us to obtain profiling information such as input/output data types and perform branch profiling.

We use this information for OpenCL compilation in later phases through the accelerator API shown in Figure 1. When the input function is executed multiple times, the AST interpreter sends the AST for compilation via partial evaluation to the Graal compiler.

2.3 OpenCL Compilation

The compilation is performed via Graal. Graal takes the specialized AST and transforms it to an intermediate representation (Graal IR) that is used for compiler optimizations.

We apply a set of OpenCL compilation phases over the Graal IR that basically removes redundant checks in the interpreted code, such as the type checks of the input data. Removing these checks when executing on GPUs is totally safe because we need to marshal and transfer the data from the CPU to the GPU and, if there is any violation, we fall back the execution to the default AST interpreter with no GPU support. Therefore we do not change the semantic of the original program. Once we apply the input checks elimination in the Graal IR, we generate the OpenCL C code and we compile it using the OpenCL runtime to the target GPU.

3. Preliminary Results

We evaluate our OpenCL JIT compiler for Ruby and R using an AMD R9 GPU. We compare our compiler approach against the CRuby and GNU-R, the Truffle versions (JRuby and FastR) and the native OpenCL C++ version.

Figure 2 shows the speedups we obtain for each version for the `daxpy` application. Our compiler approach is 20x times faster than CRuby and 500x times faster than GNU-R. Comparing to the Truffle versions, our compiler approach is 10x times faster than JRuby and 150x times faster than FastR . Comparing to the native version, our compiler approach is 1.8x slower than OpenCL C++ for each language. Prior results (Fumero et al.) show speedups up to 150x for FastR over a large set of complex applications. The Ruby implementation for GPUs is still work in progress and once this is completed, we expect to obtain similar speedups to FastR with GPU support.

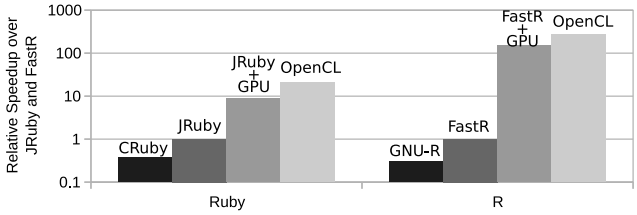

Figure 2. Speedup of the `daxpy` application implemented in Ruby and R normalized to JRuby and FastR. The higher the better.

References

G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. Graal IR: An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. VMIL 2013.

J. Fumero, M. Steuwer, L. Stadler, and C. Dubach. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. VEE 2017.

L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing R Language Execution via Aggressive Speculation. DLS 2016.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. Onward! 2013.